加微信：642945106 发送“赠送”领取赠送精品课程
发数字“2”获取众筹列表

14 | Controller选举是怎么实现的?

2020-05-21 胡夕

Kafka核心源码解读

进入课程 >

讲述：胡夕
时长 22:15 大小 20.39M

你好，我是胡夕。

上节课，我们学习了单线程事件队列模型处理 Controller 事件的代码。Controller 组件通过 ControllerEventManager 类构造了一个阻塞队列，同时配以专属的事件处理线程，实现了对各类 ControllerEvent 的处理。

这种设计思路既保证了多线程访问所需的线程安全，还简化了 Controller 端的代码结构，极大地提升了代码的可维护性。

今天，我们学习下 Controller 选举部分的源码。

还记得我在第 11 节课的案例中提到的“恢复大法”——删除 ZooKeeper 的 /controller 节点吗？当时，我们靠着这个“秘籍”涉险过关，既恢复了错误的集群状态，又避免了重启整个生产环境。

但你有没有想过，为什么删除 /controller 节点能够令集群元数据重新保持同步呢？如果不了解这背后的原理，我们是不敢贸然在生产环境做这种操作的。今天，我们要学习的就是这背后的一整套实现逻辑，重点关注下 Controller 是怎么被选举出来的。

我始终认为，只有掌握了这些知识，才算真正入门 Kafka 服务器端的代码了。作为 Broker 端最重要的组件之一，Controller 在 Kafka 中的地位无可替代。整个 Kafka 集群就只有一个 Controller，从某种意义上来说，它是目前 Kafka 这个分布式系统中唯一的“单点”。

因此，了解这个“单点”的选举触发场景，以及如何被选举出来的，对于我们后面深入理解 Controller 在集群中的作用非常有帮助。毕竟，Controller 对外提供的一些服务也是采用了类似的实现原理。

概览

ZooKeeper /controller 节点

再次强调下，**在一个 Kafka 集群中，某段时间内只能有一台 Broker 被选举为 Controller。随着时间的推移，可能会有不同的 Broker 陆续担任过 Controller 的角色，但是在某一时刻，Controller 只能由一个 Broker 担任。**

那选择哪个 Broker 充当 Controller 呢？当前，Controller 的选举过程依赖 ZooKeeper 完成。ZooKeeper 除了扮演集群元数据的“真理之源”角色，还定义了 /controller 临时节点（Ephemeral Node），以协助完成 Controller 的选举。

下面这段代码展示的是一个双 Broker 的 Kafka 集群上的 ZooKeeper 中 /controller 节点：

复制代码

```
{"version":1,"brokerid":0,"timestamp":"1585098432431"}
cZxid = 0x1a
ctime = Wed Mar 25 09:07:12 CST 2020
mZxid = 0x1a
mtime = Wed Mar 25 09:07:12 CST 2020
```

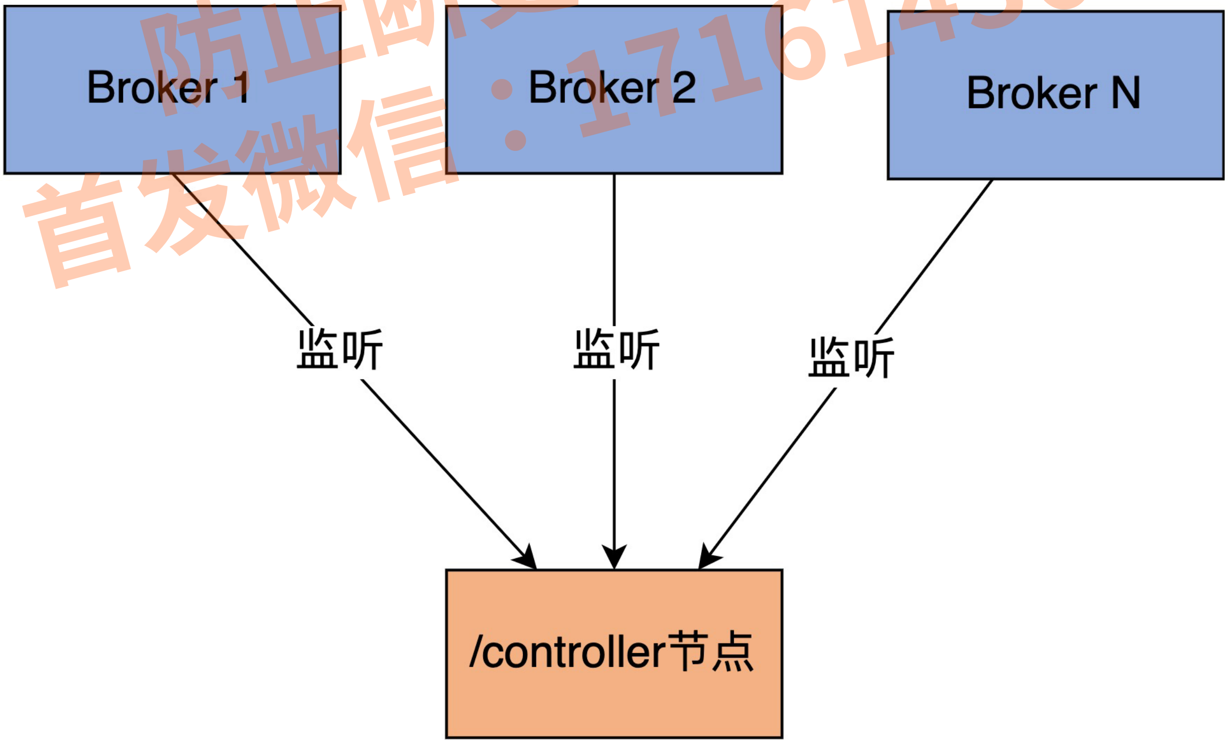
```
pZxid = 0x1a
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x100002d3a1f0000
dataLength = 54
numChildren = 0
```

有两个地方的内容，你要重点关注一下。

- Controller Broker Id 是 0，表示序号为 0 的 Broker 是集群 Controller。
- ephemeralOwner 字段不是 0x0，说明这是一个临时节点。

既然是临时节点，那么，一旦 Broker 与 ZooKeeper 的会话终止，该节点就会消失。Controller 选举就依靠了这个特性。每个 Broker 都会监听 /controller 节点随时准备应聘 Controller 角色。下图展示了 Broker 与 /controller 节点的交互关系：

防止断更 请务必加首发微信：17161443665



极客时间

如图所示，集群上所有的 Broker 都在实时监听 ZooKeeper 上的这个节点。这里的“监听”有两个含义。

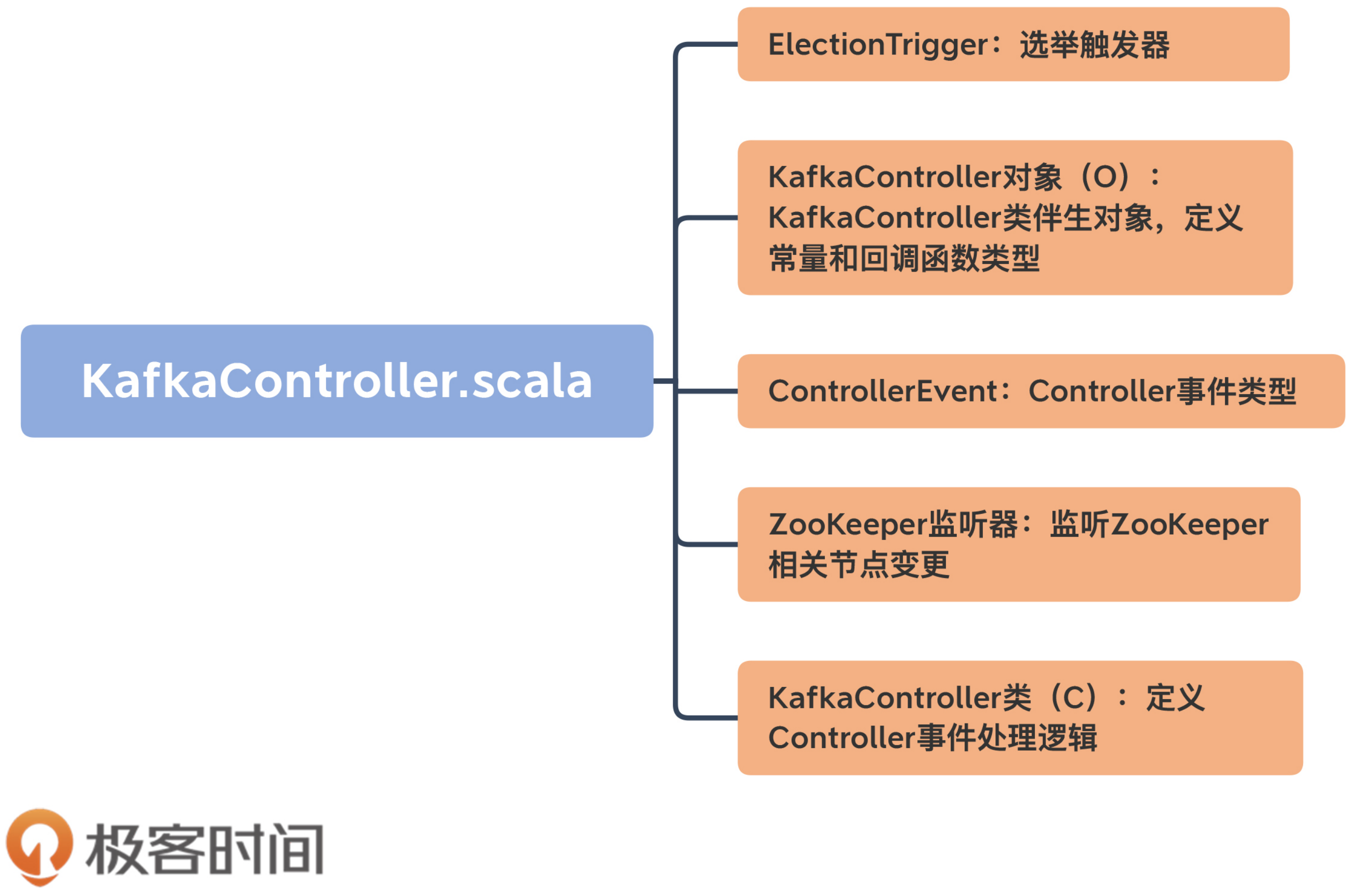
监听这个节点是否存在。倘若发现这个节点不存在，Broker 会立即“抢注”该节点，即创建 /controller 节点。创建成功的那个 Broker，即当选为新一届的 Controller。

监听这个节点数据是否发生了变更。同样，一旦发现该节点的内容发生了变化，Broker 也会立即启动新一轮的 Controller 选举。

掌握了这些基础之后，下面我们来阅读具体的源码文件：KafkaController.scala。这是一个 2200 行的大文件。我先向你介绍一下这个文件的大致结构，以免你陷入到一些繁枝末节中。

源码结构

KafkaController 文件的代码结构如下图所示：



整体而言，该文件大致由五部分组成。

选举触发器（ElectionTrigger）：这里的选举不是指 Controller 选举，而是指主题分区副本的选举，即为哪些分区选择 Leader 副本。后面在学习副本管理器和分区管理器时，我们会讲到它。

KafkaController Object：KafkaController 伴生对象，仅仅定义了一些常量和回调函数类型。

ControllerEvent：定义 Controller 事件类型。上节课我们详细学习过 Controller 事件以及基于事件的单线程事件队列模型。这部分的代码看着很多，但实际上都是千篇一律的。你看懂了一个事件的定义，其他的也就不在话下了。

各种 ZooKeeper 监听器：定义 ZooKeeper 监听器，去监听 ZooKeeper 中各个节点的变更。今天，我们重点关注监听 /controller 节点的那个监听器。

KafkaController Class：定义 KafkaController 类以及实际的处理逻辑。这是我们今天的重点学习对象。

接下来，我会给你重点介绍 KafkaController 类、ZooKeeper 监听器和 Controller 选举这三大部分。在众多的 ZooKeeper 监听器中，我会详细介绍监听 Controller 变更的监听器，它也是我们了解 Controller 选举流程的核心环节。

KafkaController 类

这个类大约有 1900 行代码，里面定义了非常多的变量和方法。这些方法大多是处理不同 Controller 事件的。后面讲到选举流程的时候，我会挑一些有代表性的来介绍。我希望你能举一反三，借此吃透其他方法的代码。毕竟，它们做的事情大同小异，至少代码风格非常相似。

在学习重要的方法之前，我们必须要先掌握 KafkaController 类的定义。接下来，我们从 4 个维度来进行学习，分别是原生字段、辅助字段、各类 ZooKeeper 监听器字段和统计字段。

弄明白了这些字段的含义之后，再去看操作这些字段的方法，会更加有的放矢，理解起来也会更加容易。

原生字段

首先来看原生字段。所谓的原生字段，是指在创建一个 KafkaController 实例时，需要指定的字段。

先来看下 KafkaController 类的定义代码：

复制代码

```
// 字段含义:
// config: Kafka配置信息，通过它，你能拿到Broker端所有参数的值
// zkClient: ZooKeeper客户端，Controller与ZooKeeper的所有交互均通过该属性完成
// time: 提供时间服务(如获取当前时间)的工具类
// metrics: 实现指标监控服务(如创建监控指标)的工具类
// initialBrokerInfo: Broker节点信息，包括主机名、端口号，所用监听器等
// initialBrokerEpoch: Broker Epoch值，用于隔离老Controller发送的请求
// tokenManager: 实现Delegation token管理的工具类。Delegation token是一种轻量级的认证
// threadNamePrefix: Controller端事件处理线程名字前缀
class KafkaController(val config: KafkaConfig,
                      zkClient: KafkaZkClient,
                      time: Time,
                      metrics: Metrics,
                      initialBrokerInfo: BrokerInfo,
                      initialBrokerEpoch: Long,
                      tokenManager: DelegationTokenManager,
                      threadNamePrefix: Option[String] = None)
  extends ControllerEventProcessor with Logging with KafkaMetricsGroup {
  ......
}
```

就像我上节课说过的，KafkaController 实现了 ControllerEventProcessor 接口，因而也就实现了处理 Controller 事件的 process 方法。这里面比较重要的字段有 3 个。

config：KafkaConfig 类实例，里面封装了 Broker 端所有参数的值。

zkClient：ZooKeeper 客户端类，定义了与 ZooKeeper 交互的所有方法。

initialBrokerEpoch：Controller 所在 Broker 的 Epoch 值。Kafka 使用它来确保 Broker 不会处理老 Controller 发来的请求。

其他字段要么是像 time、metrics 一样，是工具类字段，要么是像 initialBrokerInfo、tokenManager 字段一样，使用场景很有限，我就不展开讲了。

辅助字段

除了原生字段之外，KafkaController 还定义了很多辅助字段，帮助实现 Controller 的各类功能。

我们来看一些重要的辅助字段：

复制代码

```
......
// 集群元数据类，保存集群所有元数据
val controllerContext = new ControllerContext
// Controller端通道管理器类，负责Controller向Broker发送请求
var controllerChannelManager = new ControllerChannelManager(controllerContext,
  stateChangeLogger, threadNamePrefix)
// 线程调度器，当前唯一负责定期执行Leader重选举
private[controller] val kafkaScheduler = new KafkaScheduler(1)
// Controller事件管理器，负责管理事件处理线程
private[controller] val eventManager = new ControllerEventManager(config.broke
  controllerContext.stats.rateAndTimeMetrics)
......
// 副本状态机，负责副本状态转换
val replicaStateMachine: ReplicaStateMachine = new ZkReplicaStateMachine(confi
  new ControllerBrokerRequestBatch(config, controllerChannelManager, eventMana
// 分区状态机，负责分区状态转换
val partitionStateMachine: PartitionStateMachine = new ZkPartitionStateMachine
  new ControllerBrokerRequestBatch(config, controllerChannelManager, eventMana
// 主题删除管理器，负责删除主题及日志
val topicDeletionManager = new TopicDeletionManager(config, controllerContext,
  partitionStateMachine, new ControllerDeletionClient(this, zkClient))
......
```

其中，有 7 个字段是重中之重。

controllerContext：集群元数据类，保存集群所有元数据。

controllerChannelManager：Controller 端通道管理器类，负责 Controller 向 Broker 发送请求。

kafkaScheduler：线程调度器，当前唯一负责定期执行分区重平衡 Leader 选举。

eventManager：Controller 事件管理器，负责管理事件处理线程。

replicaStateMachine：副本状态机，负责副本状态转换。

partitionStateMachine：分区状态机，负责分区状态转换。

topicDeletionManager：主题删除管理器，负责删除主题及日志。

各类 ZooKeeper 监听器

我们今天开头学到的 ControllerChangeHandler 仅仅是其中的一种。实际上，该类定义了很多监听器，如下所示：

复制代码

```
// Controller节点ZooKeeper监听器
private val controllerChangeHandler = new ControllerChangeHandler(eventManager
// Broker数量ZooKeeper监听器
private val brokerChangeHandler = new BrokerChangeHandler(eventManager)
// Broker信息变更ZooKeeper监听器集合
private val brokerModificationsHandlers: mutable.Map[Int, BrokerModificationsH
// 主题数量ZooKeeper监听器
private val topicChangeHandler = new TopicChangeHandler(eventManager)
// 主题删除ZooKeeper监听器
private val topicDeletionHandler = new TopicDeletionHandler(eventManager)
// 主题分区变更ZooKeeper监听器
private val partitionModificationsHandlers: mutable.Map[String, PartitionModif
// 主题分区重分配ZooKeeper监听器
private val partitionReassignmentHandler = new PartitionReassignmentHandler(ev
// Preferred Leader选举ZooKeeper监听器
private val preferredReplicaElectionHandler = new PreferredReplicaElectionHand
// ISR副本集合变更ZooKeeper监听器
private val isrChangeNotificationHandler = new IsrChangeNotificationHandler(ev
// 日志路径变更ZooKeeper监听器
private val logDirEventNotificationHandler = new LogDirEventNotificationHandle
```

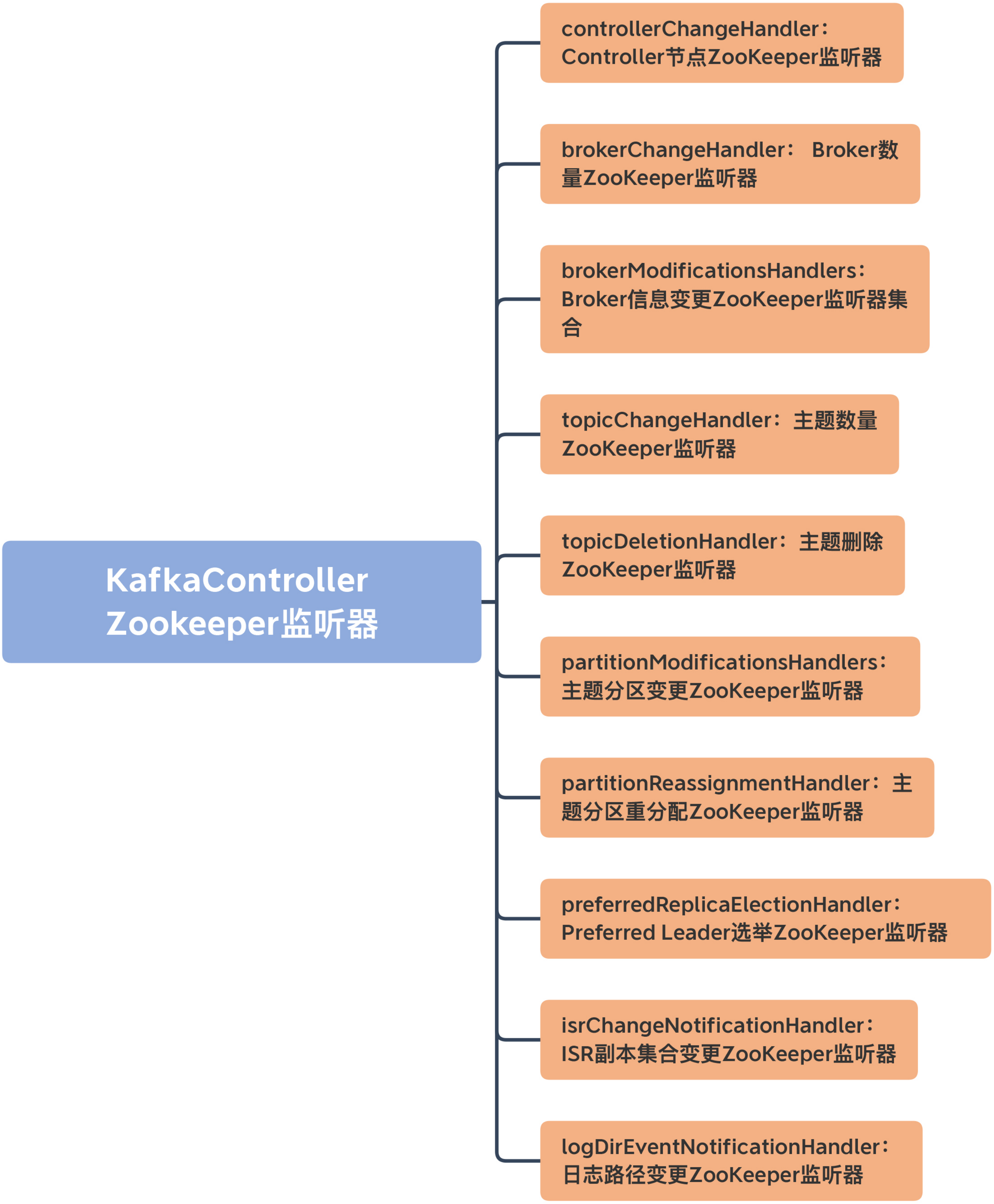
我分别解释一下这些 ZooKeeper 监听器的作用：

- **controllerChangeHandler**：前面说过，它是监听 /controller 节点变更的。这种变更包括节点创建、删除以及数据变更。
- **brokerChangeHandler**：监听 Broker 的数量变化。
- **brokerModificationsHandlers**：监听 Broker 的数据变更，比如 Broker 的配置信息发生的变化。
- **topicChangeHandler**：监控主题数量变更。
- **topicDeletionHandler**：监听主题删除节点 /admin/delete_topics 的子节点数量变更。
- **partitionModificationsHandlers**：监控主题分区数据变更的监听器，比如，新增加了副本、分区更换了 Leader 副本。
- **partitionReassignmentHandler**：监听分区副本重分配任务。一旦发现新提交的任务，就为目标分区执行副本重分配。
- **preferredReplicaElectionHandler**：监听 Preferred Leader 选举任务。一旦发现新提交的任务，就为目标主题执行 Preferred Leader 选举。

isrChangeNotificationHandler：监听 ISR 副本集合变更。一旦被触发，就需要获取 ISR 发生变更的分区列表，然后更新 Controller 端对应的 Leader 和 ISR 缓存元数据。

logDirEventNotificationHandler：监听日志路径变更。一旦被触发，需要获取受影响的 Broker 列表，然后处理这些 Broker 上失效的日志路径。

我画了一张脑图，希望可以帮助你更高效地记住这些 ZooKeeper 监听器：



极客时间

统计字段

最后，我们来看统计字段。

这些统计字段大多用于计算统计指标。有的监控指标甚至是非常重要的 Controller 监控项，比如 ActiveControllerCount 指标。下面，我们来了解下 KafkaController 都定义了哪些统计字段。这些指标的含义一目了然，非常清晰，我用注释的方式给出每个字段的含义：

复制代码

```
// 当前Controller所在Broker Id
@volatile private var activeControllerId = -1
// 离线分区总数
@volatile private var offlinePartitionCount = 0
// 满足Preferred Leader选举条件的总分区数
@volatile private var preferredReplicaImbalanceCount = 0
// 总主题数
@volatile private var globalTopicCount = 0
// 总主题分区数
@volatile private var globalPartitionCount = 0
// 待删除主题数
@volatile private var topicsToDeleteCount = 0
//待删除副本数
@volatile private var replicasToDeleteCount = 0
// 暂时无法删除的主题数
@volatile private var ineligibleTopicsToDeleteCount = 0
// 暂时无法删除的副本数
@volatile private var ineligibleReplicasToDeleteCount = 0
```

好了，KafkaController 类的定义我们就全部介绍完了。再次强调下，因为 KafkaController 类的代码很多，我强烈建议你熟练掌握这些字段的含义，因为后面的所有方法都是围绕着这些字段进行操作的。

接下来，我以 Controller 的选举流程为例，引出 KafkaController 的一些方法的实现原理。不过，在此之前，我们要学习监听 Controller 变更的 ZooKeeper 监听器：ControllerChangeHandler 的源码。

ControllerChangeHandler 监听器

就像我前面说到的，KafkaController 定义了十几种 ZooKeeper 监听器。和 Controller 相关的监听器是 ControllerChangeHandler，用于监听 Controller 的变更，定义代码如下：

复制代码

```
class ControllerChangeHandler(eventManager: ControllerEventManager) extends ZN
  // ZooKeeper中Controller节点路径，即/controller
```

```
  override val path: String = ControllerZNode.path
  // 监听/controller节点创建事件
  override def handleCreation(): Unit = eventManager.put(ControllerChange)
  // 监听/controller节点被删除事件
  override def handleDeletion(): Unit = eventManager.put(Reelect)
  // 监听/controller节点数据变更事件
  override def handleDataChange(): Unit = eventManager.put(ControllerChange)
}
```

该监听器接收 ControllerEventManager 实例，实现了 ZNodeChangeHandler 接口的三个方法：**handleCreation**、**handleDeletion** 和 **handleDataChange**。该监听器下的 path 变量，实际上就是 /controller 字符串，表示它监听 ZooKeeper 的这个节点。

3 个 handle 方法都用于监听 /controller 节点的变更，但实现细节上稍有不同。

handleCreation 和 handleDataChange 的处理方式是向事件队列写入 ControllerChange 事件；handleDeletion 的处理方式是向事件队列写入 Reelect 事件。

Deletion 表明 ZooKeeper 中 /controller 节点不存在了，即 Kafka 集群中的 Controller 暂时空缺了。因为它和 Creation 和 DataChange 是不同的状态，需要区别对待，因此，Reelect 事件做的事情要比 ControllerChange 的多：处理 ControllerChange 事件，只需要当前 Broker 执行“卸任 Controller”的逻辑即可，而 Reelect 事件是重选举，除了 Broker 执行卸任逻辑之外，还要求 Broker 参与到重选举中来。

由于 KafkaController 的 process 方法代码非常长，因此，我节选了刚刚提到的那两个事件的处理代码：

复制代码

```
// process方法(部分)
override def process(event: ControllerEvent): Unit = {
    try {
      event match {
       ......
       // ControllerChange事件
       case ControllerChange =>
          processControllerChange()
       // Reelect事件
       case Reelect =>
          processReelect()
        ......

```

```
        }
      }
      ......
  }
  // 如果是ControllerChange事件，仅执行卸任逻辑即可
  private def processControllerChange(): Unit = {
      maybeResign()
    }
  // 如果是Reelect事件，还需要执行elect方法参与新一轮的选举
  private def processReelect(): Unit = {
      maybeResign()
      elect()
```

可以看到，虽然代码非常长，但整体结构却工整清晰，全部都是基于模式匹配的事件处理。process 方法会根据给定的 Controller 事件类型，调用对应的 process*** 方法处理该事件。这里只列举了 ZooKeeper 端 /controller 节点监听器监听的两类事件，以及对应的处理方法。

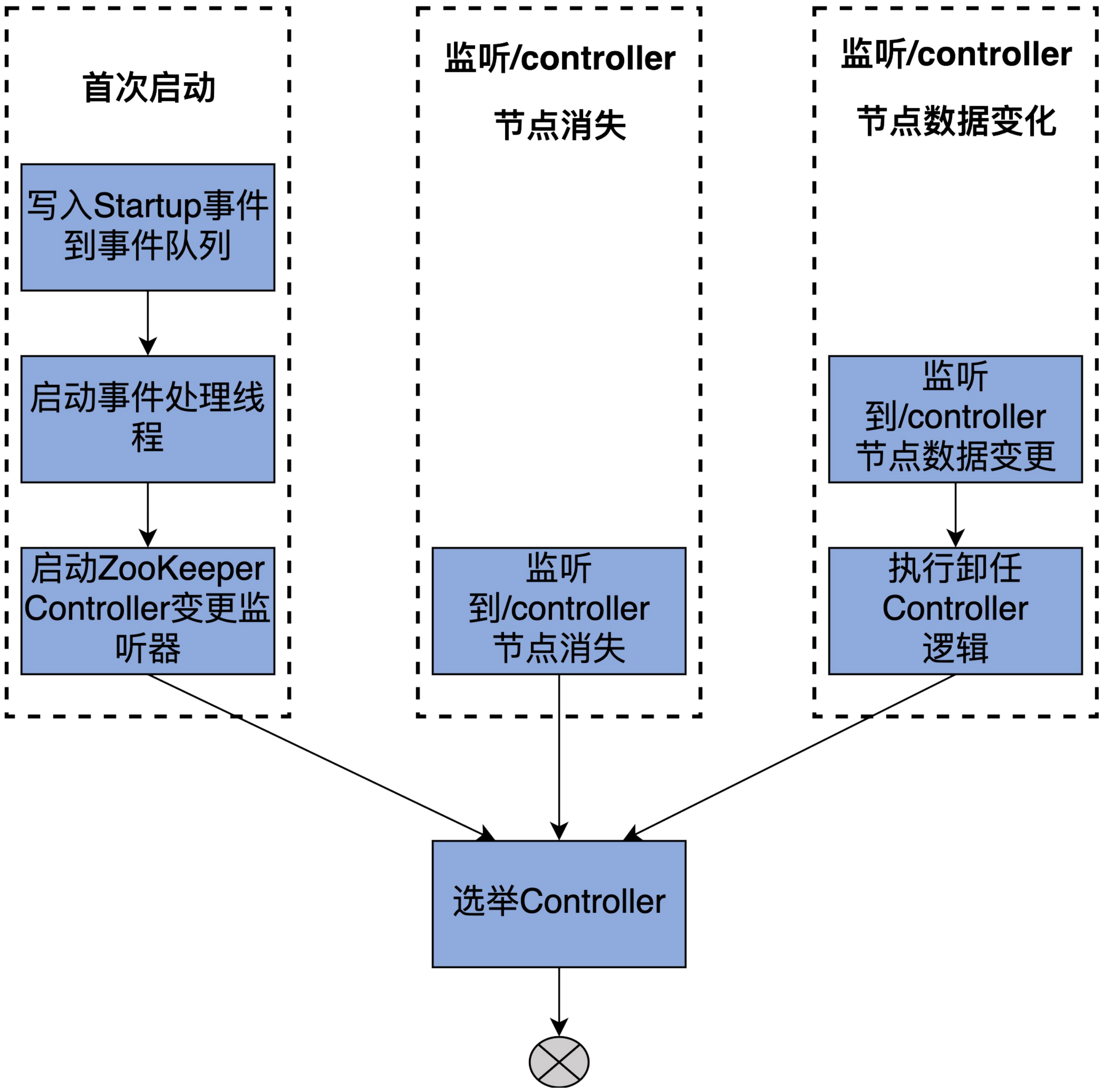
对于 ControllerChange 事件而言，处理方式是调用 maybeResign 去执行 Controller 的卸任逻辑。如果是 Reelect 事件，除了执行卸任逻辑之外，还要额外执行 elect 方法进行新一轮的 Controller 选举。

Controller 选举流程

说完了 ControllerChangeHandler 源码，我们来看下 Controller 的选举。所谓的 Controller 选举，是指 Kafka 选择集群中一台 Broker 行使 Controller 职责。整个选举过程分为两个步骤：触发选举和开始选举。

触发选举

我先用一张图展示下可能触发 Controller 选举的三个场景。



极客时间

这三个场景是：

1. 集群从零启动时；
2. Broker 侦测 /controller 节点消失时；
3. Broker 侦测到 /controller 节点数据发生变更时。

这三个场景殊途同归，最后都要执行选举 Controller 的动作。我来一一解释下这三个场景，然后再介绍选举 Controller 的具体操作。

场景一：集群从零启动

集群首次启动时，Controller 尚未被选举出来。于是，Broker 启动后，首先将 Startup 这个 ControllerEvent 写入到事件队列中，然后启动对应的事件处理线程和 ControllerChangeHandler ZooKeeper 监听器，最后依赖事件处理线程进行 Controller 的选举。

在源码中，KafkaController 类的 startup 方法就是做这些事情的。当 Broker 启动时，它会调用这个方法启动 ControllerEventThread 线程。值得注意的是，**每个 Broker 都需要做这些事情，不是说只有 Controller 所在的 Broker 才需要执行这些逻辑**。

startup 方法的主体代码如下：

复制代码

```
def startup() = {
  // 第1步：注册ZooKeeper状态变更监听器，它是用于监听Zookeeper会话过期的
  zkClient.registerStateChangeHandler(new StateChangeHandler {
    override val name: String = StateChangeHandlers.ControllerHandler
    override def afterInitializingSession(): Unit = {
      eventManager.put(RegisterBrokerAndReelect)
    }
    override def beforeInitializingSession(): Unit = {
      val queuedEvent = eventManager.clearAndPut(Expire)
      queuedEvent.awaitProcessing()
    }
  })
  // 第2步：写入Startup事件到事件队列
  eventManager.put(Startup)
  // 第3步：启动ControllerEventThread线程，开始处理事件队列中的ControllerEvent
  eventManager.start()
}

```

首先，startup 方法会注册 ZooKeeper 状态变更监听器，用于监听 Broker 与 ZooKeeper 之间的会话是否过期。接着，写入 Startup 事件到事件队列，然后启动 ControllerEventThread 线程，开始处理事件队列中的 Startup 事件。

接下来，我们来学习下 KafkaController 的 process 方法处理 Startup 事件的方法：

复制代码

```
// KafkaController的process方法，
override def process(event: ControllerEvent): Unit = {
```

```
    try {
      event match {
       ......
       case Startup =>
          processStartup() // 处理Startup事件
      }
    }
    ......
}
private def processStartup(): Unit = {
   // 注册ControllerChangeHandler ZooKeeper监听器
   zkClient.registerZNodeChangeHandlerAndCheckExistence(
    controllerChangeHandler)
   // 执行Controller选举
   elect()

```

从这段代码可知，process 方法调用 processStartup 方法去处理 Startup 事件。而 processStartup 方法又会调用 zkClient 的 registerZNodeChangeHandlerAndCheckExistence 方法注册 ControllerChangeHandler 监听器。

值得注意的是，虽然前面的三个场景是并列的关系，但实际上，后面的两个场景必须要等场景一的这一步成功执行之后，才能被触发。

这三种场景都要选举 Controller，因此，我们最后统一学习 elect 方法的代码实现。

总体来说，集群启动时，Broker 通过向事件队列“塞入”Startup 事件的方式，来触发 Controller 的竞选。

场景二：/controller 节点消失

Broker 检测到 /controller 节点消失时，就意味着，此时整个集群中没有 Controller。因此，所有检测到 /controller 节点消失的 Broker，都会立即调用 elect 方法执行竞选逻辑。

你可能会问：“Broker 是怎么侦测到 ZooKeeper 上的这一变化的呢？”实际上，这是 ZooKeeper 监听器提供的功能，换句话说，这是 Apache ZooKeeper 自己实现的功能，所以我们才说，Kafka 依赖 ZooKeeper 完成 Controller 的选举。

讲到这里，我说点题外话，社区最近正在酝酿彻底移除 ZooKeeper 依赖。具体到 Controller 端的变化，就是在 Kafka 内部实现一个类似于 Raft 的共识算法来选举 Controller。我会在后面的特别放送里详细讲一下社区移除 ZooKeeper 的全盘计划。

场景三：/controller 节点数据变更

Broker 检测到 /controller 节点数据发生变化，通常表明，Controller “易主”了，这就分为两种情况：

- 如果 Broker 之前是 Controller，那么该 Broker 需要首先执行卸任操作，然后再尝试竞选；
- 如果 Broker 之前不是 Controller，那么，该 Broker 直接去竞选新 Controller。

具体到代码层面，maybeResign 方法形象地说明了这两种情况。你要注意方法中的 maybe 字样，这表明，Broker 可能需要执行卸任操作，也可能不需要。Kafka 源码非常喜欢用 maybe*** 来命名方法名，以表示那些在特定条件下才需要执行的逻辑。以下是 maybeResign 的实现：

复制代码

```
1  private def maybeResign(): Unit = {
2    // 非常关键的一步！这是判断是否需要执行卸任逻辑的重要依据！
3    // 判断该Broker之前是否是Controller
4    val wasActiveBeforeChange = isActive
5    // 注册ControllerChangeHandler监听器
6    zkClient.registerZNodeChangeHandlerAndCheckExistence(
7      controllerChangeHandler)
8    // 获取当前集群Controller所在的Broker Id，如果没有Controller则返回-1
9    activeControllerId = zkClient.getControllerId.getOrElse(-1)
10   // 如果该Broker之前是Controller但现在不是了
11   if (wasActiveBeforeChange && !isActive) {
12     onControllerResignation() // 执行卸任逻辑
13   }
14 }
```

代码的第一行非常关键，它是决定是否需要执行卸任的重要依据。毕竟，如果 Broker 之前不是 Controller，那何来“卸任”一说呢？之后代码要注册 ControllerChangeHandler 监听器，获取当前集群 Controller 所在的 Broker ID，如果没有 Controller，则返回 -1。有了这些数据之后，maybeResign 方法需要判断该 Broker 是否之前是 Controller 但现在不

是了。如果是这种情况的话，则调用 onControllerResignation 方法执行 Controller 卸任逻辑。

说到“卸任”，你可能会问：“卸任逻辑是由哪个方法执行的呢？”实际上，这是由 onControllerResignation 方法执行的，它主要是用于清空各种数据结构的值、取消 ZooKeeper 监听器、关闭各种状态机以及管理器，等等。我用注释的方式给出它的逻辑实现：

复制代码

```
private def onControllerResignation(): Unit = {
  debug("Resigning")
  // 取消ZooKeeper监听器的注册
  zkClient.unregisterZNodeChildChangeHandler(
    isrChangeNotificationHandler.path)
  zkClient.unregisterZNodeChangeHandler(
    partitionReassignmentHandler.path)
  zkClient.unregisterZNodeChangeHandler(
    preferredReplicaElectionHandler.path)
  zkClient.unregisterZNodeChildChangeHandler(
    logDirEventNotificationHandler.path)
  unregisterBrokerModificationsHandler(
    brokerModificationsHandlers.keySet)
  // 关闭Kafka线程调度器，其实就是取消定期的Leader重选举
  kafkaScheduler.shutdown()
  // 将统计字段全部清0
  offlinePartitionCount = 0
  preferredReplicaImbalanceCount = 0
  globalTopicCount = 0
  globalPartitionCount = 0
  topicsToDeleteCount = 0
  replicasToDeleteCount = 0
  ineligibleTopicsToDeleteCount = 0
  ineligibleReplicasToDeleteCount = 0
  // 关闭Token过期检查调度器
  if (tokenCleanScheduler.isStarted)
    tokenCleanScheduler.shutdown()
  // 取消分区重分配监听器的注册
  unregisterPartitionReassignmentIsrChangeHandlers()
  // 关闭分区状态机
  partitionStateMachine.shutdown()
  // 取消主题变更监听器的注册
  zkClient.unregisterZNodeChildChangeHandler(topicChangeHandler.path)
  // 取消分区变更监听器的注册
  unregisterPartitionModificationsHandlers(
    partitionModificationsHandlers.keys.toSeq)
  // 取消主题删除监听器的注册
  zkClient.unregisterZNodeChildChangeHandler(
```

```
    topicDeletionHandler.path)
  // 关闭副本状态机
  replicaStateMachine.shutdown()
  // 取消Broker变更监听器的注册
  zkClient.unregisterZNodeChildChangeHandler(brokerChangeHandler.path)
  // 关闭Controller通道管理器
  controllerChannelManager.shutdown()
  // 清空集群元数据
  controllerContext.resetContext()
  info("Resigned")
}

```

选举 Controller

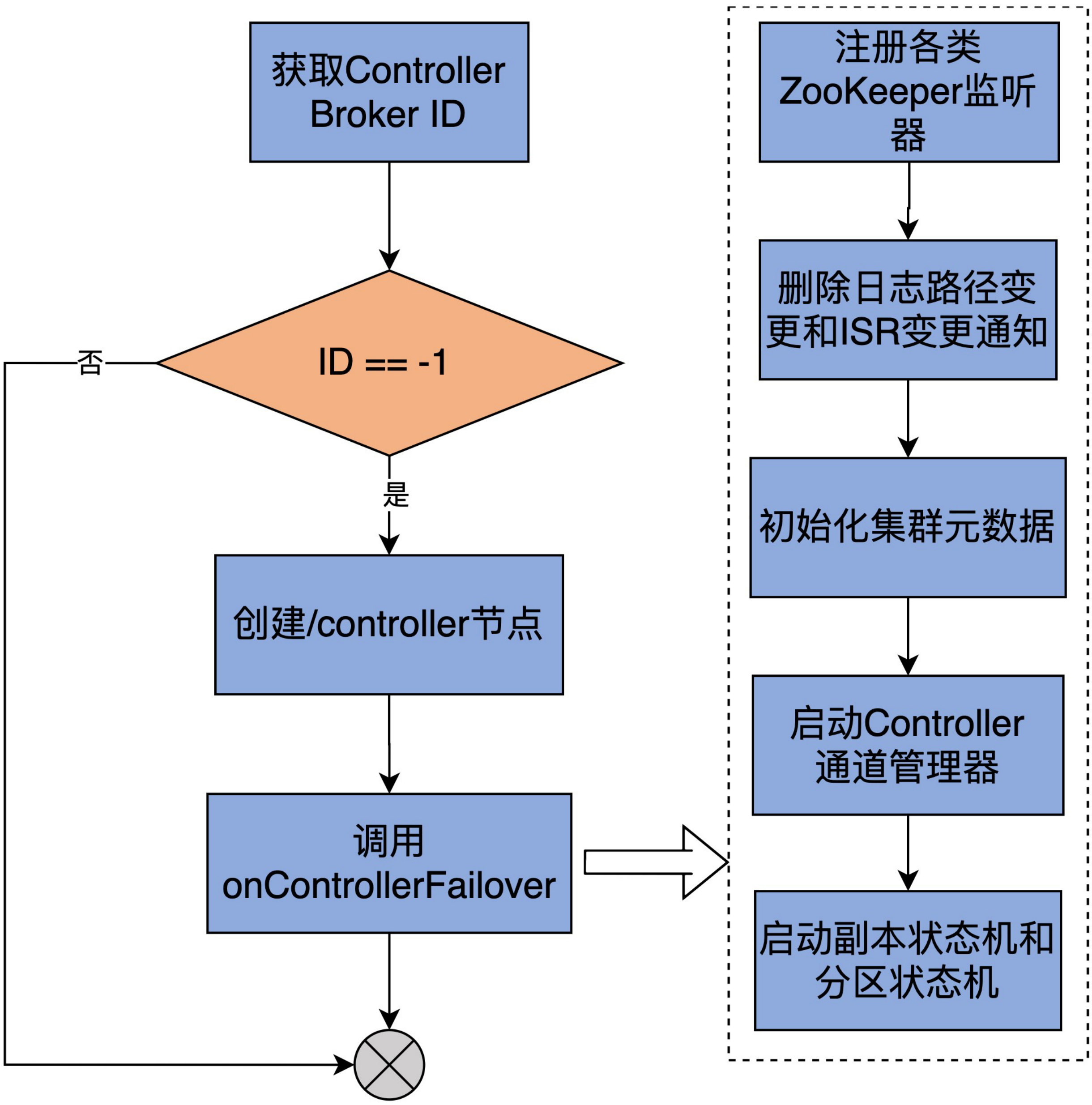
讲完了触发场景，接下来，我们就要学习 Controller 选举的源码了。前面说过了，这三种选举场景最后都会调用 elect 方法来执行选举逻辑。我们来看下它的实现：

复制代码

```
private def elect(): Unit = {
    // 第1步：获取当前Controller所在Broker的序号，如果Controller不存在，显式标记为-1
    activeControllerId = zkClient.getControllerId.getOrElse(-1)

    // 第2步：如果当前Controller已经选出来了，直接返回即可
    if (activeControllerId != -1) {
      debug(s"Broker $activeControllerId has been elected as the controller, s
      return
    }

    try {
      // 第3步：注册Controller相关信息
      // 主要是创建/controller节点
      val (epoch, epochZkVersion) = zkClient.registerControllerAndIncrementCon
      controllerContext.epoch = epoch
      controllerContext.epochZkVersion = epochZkVersion
      activeControllerId = config.brokerId

      info(s"${config.brokerId} successfully elected as the controller. Epoch
        s"and epoch zk version is now ${controllerContext.epochZkVersion}")

      // 第4步：执行当选Controller的后续逻辑
      onControllerFailover()
    } catch {
      case e: ControllerMovedException =>
        maybeResign()

        if (activeControllerId != -1)
          debug(s"Broker $activeControllerId was elected as controller instead
```

```
        else
          warn("A controller has been elected but just resigned, this will resu

      case t: Throwable =>
        error(s"Error while electing or becoming controller on broker ${config
          s"Trigger controller movement immediately", t)
        triggerControllerMove()
    }

```

为了帮助你更好地理解这个方法，我再画一张图来进行说明：



极客时间

该方法首先检查 Controller 是否已经选出来了。要知道，集群中的所有 Broker 都要执行这些逻辑，因此，非常有可能出现某些 Broker 在执行 elect 方法时，Controller 已经被选出来的情况。如果 Controller 已经选出来了，那么，自然也就不用再做什么了。相反地，如果 Controller 尚未被选举出来，那么，代码会尝试创建 /controller 节点去抢注 Controller。

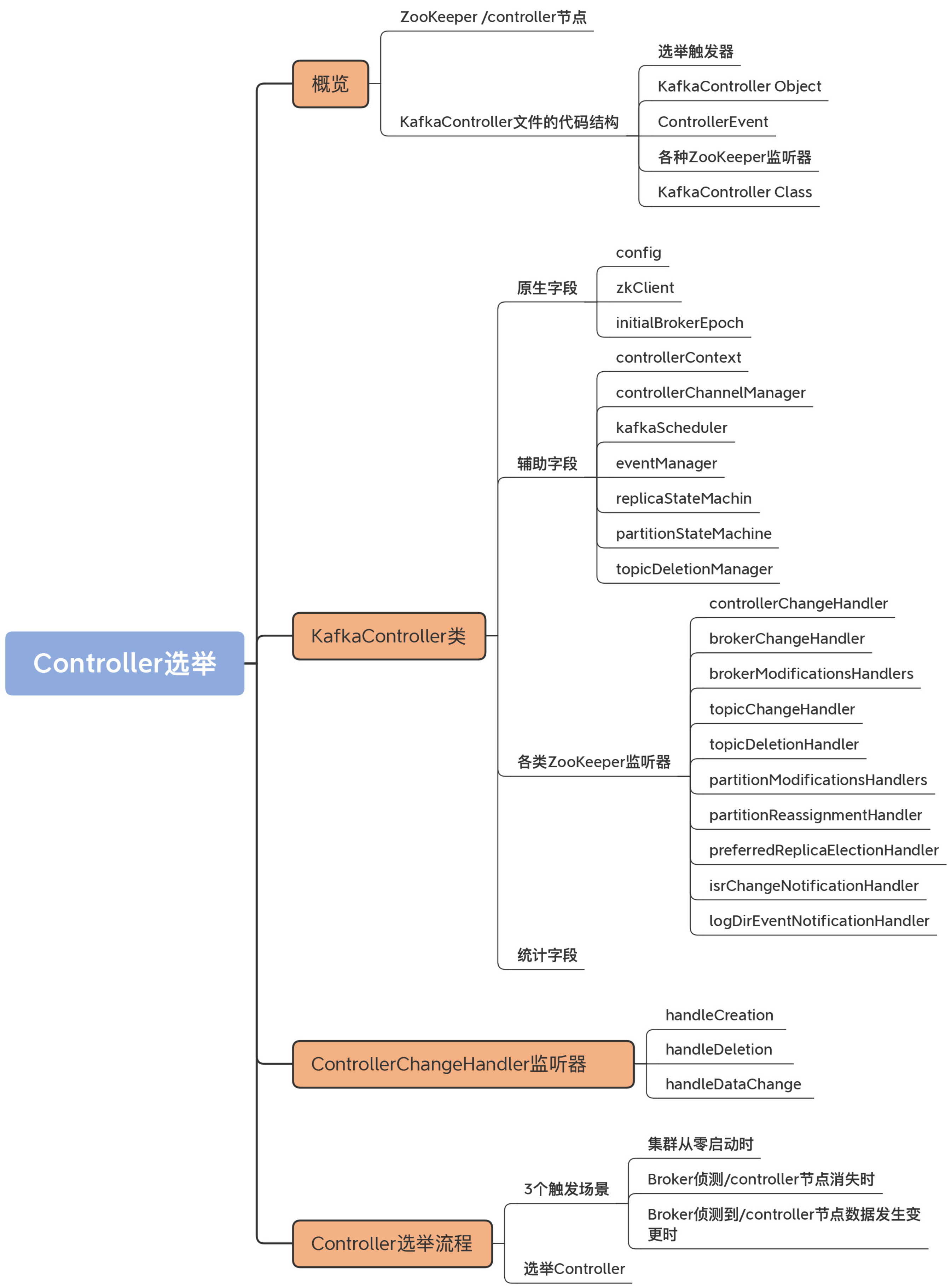
一旦抢注成功，就调用 onControllerFailover 方法，执行选举成功后的动作。这些动作包括注册各类 ZooKeeper 监听器、删除日志路径变更和 ISR 副本变更通知事件、启动 Controller 通道管理器，以及启动副本状态机和分区状态机。

如果抢注失败了，代码会抛出 ControllerMovedException 异常。这通常表明 Controller 已经被其他 Broker 抢先占据了，那么，此时代码调用 maybeResign 方法去执行卸任逻辑。

总结

今天，我们梳理了 Controller 选举的全过程，包括 Controller 如何借助 ZooKeeper 监听器实现监听 Controller 节点，以及 Controller 的选举触发场景和完整流程。我们来回顾一下这节课的重点。

- Controller 依赖 ZooKeeper 实现 Controller 选举，主要是借助于 /controller 临时节点和 ZooKeeper 的监听器机制。
- Controller 触发场景有 3 种：集群启动时；/controller 节点被删除时；/controller 节点数据变更时。
- 源码最终调用 elect 方法实现 Controller 选举。


Controller选举
概览
ZooKeeper /controller节点
KafkaController文件的代码结构
选举触发器
KafkaController Object
ControllerEvent
各种ZooKeeper监听器
KafkaController Class
KafkaController类
原生字段
config
zkClient
initialBrokerEpoch
辅助字段
controllerContext
controllerChannelManager
kafkaScheduler
eventManager
replicaStateMachin
partitionStateMachine
topicDeletionManager
各类ZooKeeper监听器
controllerChangeHandler
brokerChangeHandler
brokerModificationsHandlers
topicChangeHandler
topicDeletionHandler
partitionModificationsHandlers
partitionReassignmentHandler
preferredReplicaElectionHandler
isrChangeNotificationHandler
logDirEventNotificationHandler
统计字段
ControllerChangeHandler监听器
handleCreation
handleDeletion
handleDataChange
Controller选举流程
3个触发场景
集群从零启动时
Broker侦测/controller节点消失时
Broker侦测到/controller节点数据发生变更时
选举Controller

极客时间

下节课，我将带你学习 Controller 的其他重要功能，包括它如何管理 Broker 和副本等。你千万不要错过。

课后讨论

在这节课刚开始的时候，我提到，删除 /controller 会触发 Controller 选举，之后会同步集群元数据信息。那么，你知道源码是在哪里更新的元数据请求吗？

欢迎你在留言区畅所欲言，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | ControllerEventManager：变身单线程后的Controller如何处理事件？

下一篇 15 | 如何理解Controller在Kafka集群中的作用？

精选留言 (3)

写留言

胡夕 置顶

2020-05-29

你好，我是胡夕。我来公布上节课的“课后讨论”题答案啦～

上节课，咱们重点了解了Controller的单线程+阻塞队列消息处理模式。课后我请你思考这样一个问题：ControllerEventManager的put方法代码是否需要加putLock保护。这是一个开放式的问题，就我个人的观点，我认为是不需要的。因为queue本身是线程安全的...

展开

thomas

2020-05-28

老师，新建一个topic时，是如何决定选哪一个是领导者副本？假如有两个broker，replica. factor=2

作者回复: 一般是按照副本列表中的第一个副本是leader副本。而副本列表一般是按照brokerid顺序排列

空知

2020-05-21

调用onControllerFailover时候
sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.toSeq, Set.empty)
向集群里的brokers发送更新元数据请求

作者回复: 对，让它们去更新元数据